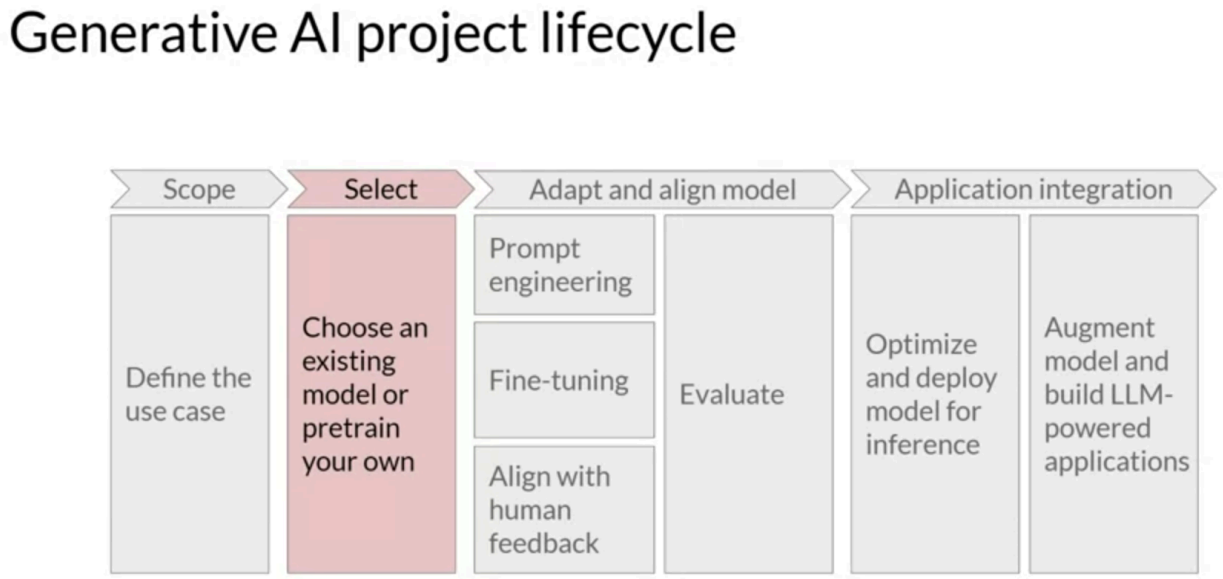
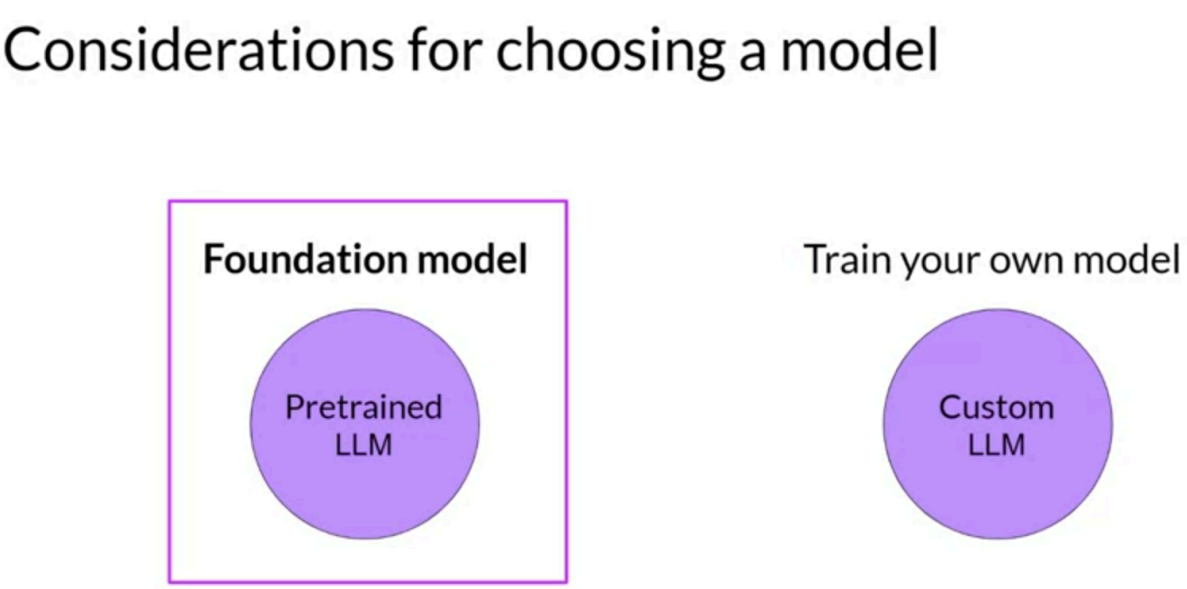
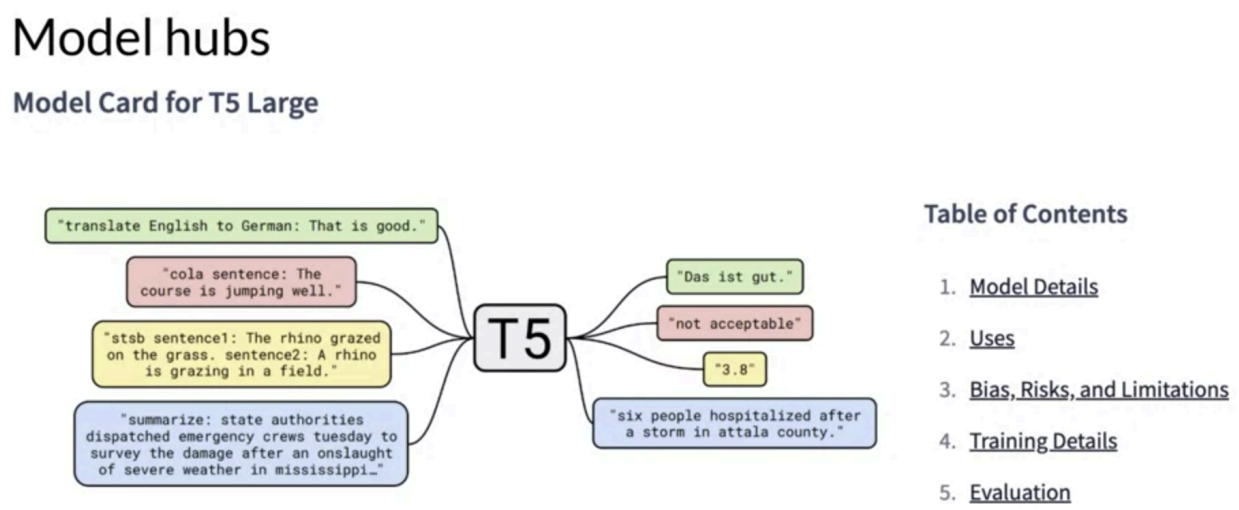
Generative-AI project lifecycle:

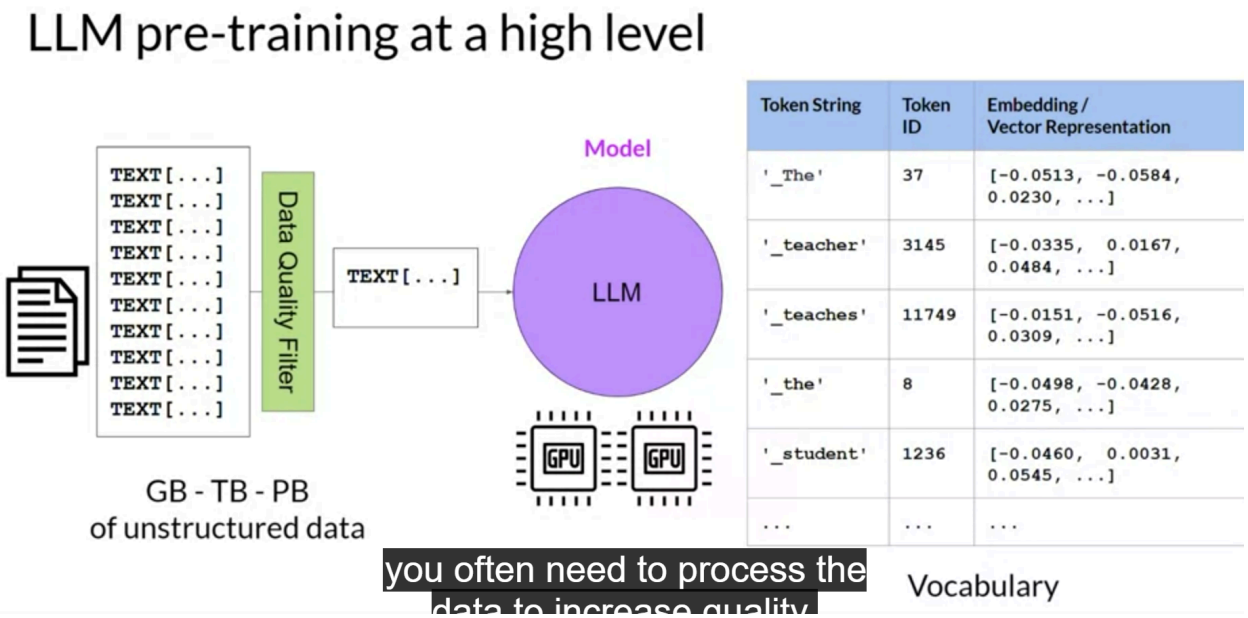


Choosing a model:





LLMs training:



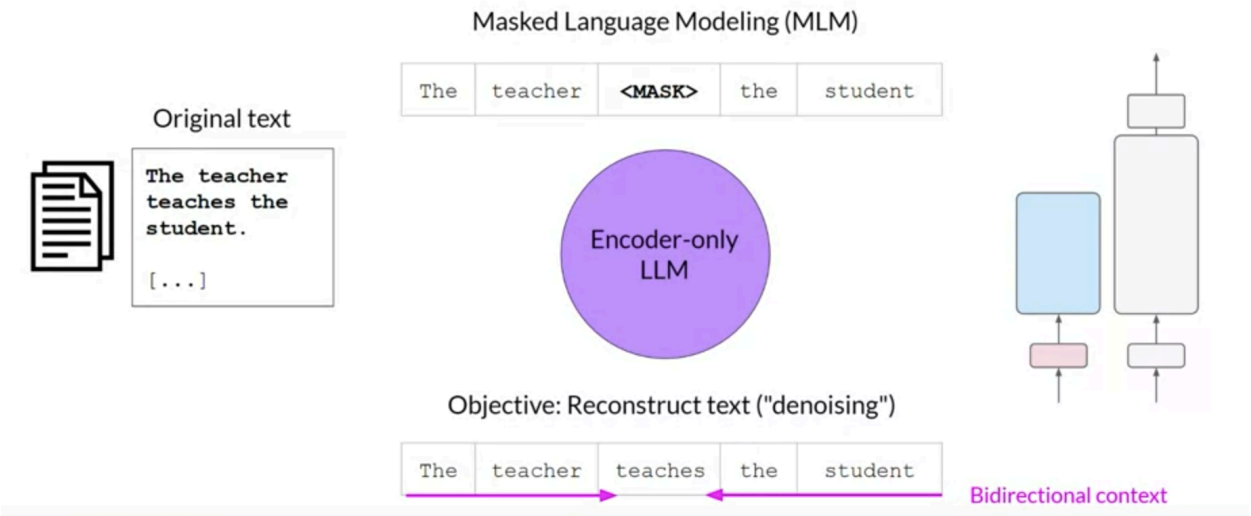
Pre-trained using scraped data or a lot of data to understand the inherent structure of the data
Encoder -> generates a contextualized vector representation of each token.

If we want to pre-train our own model, we need to curate large data.

Transformers -> Encoder-only, Encoder-decoder, decoder-only

Encoder-only -> Autoencoding models

Autoencoding models



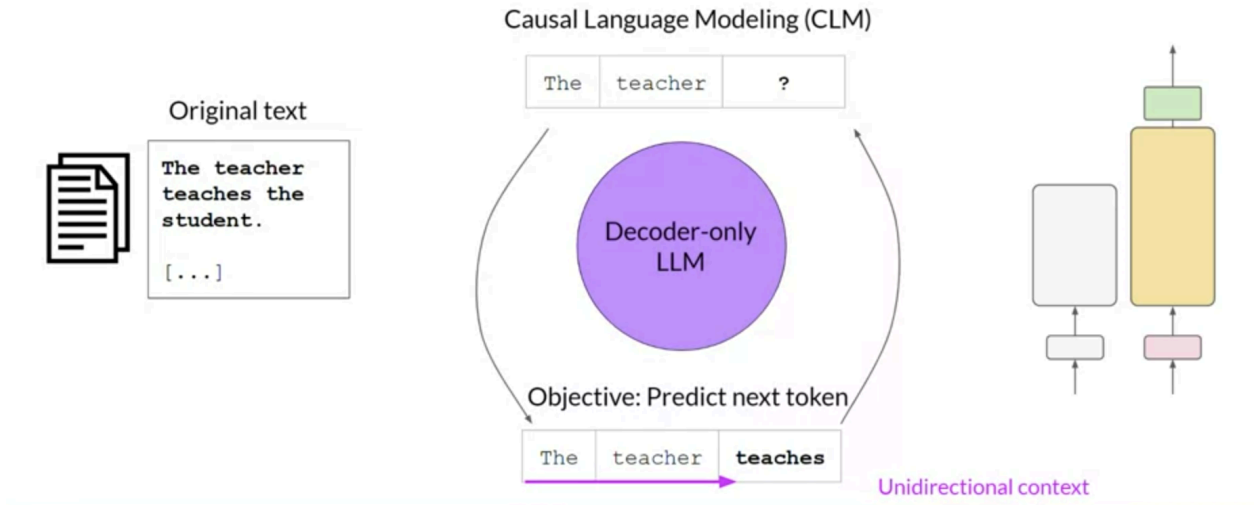
pre-trained using masked language modeling
Here tokens are masked and the encoder is required to predict the tokens using information from words that are before and after the token.

Sentence classification tasks and token classification
Sentiment analysis, Named entity recognition, or word classification

Example: BERT, ROBERTA

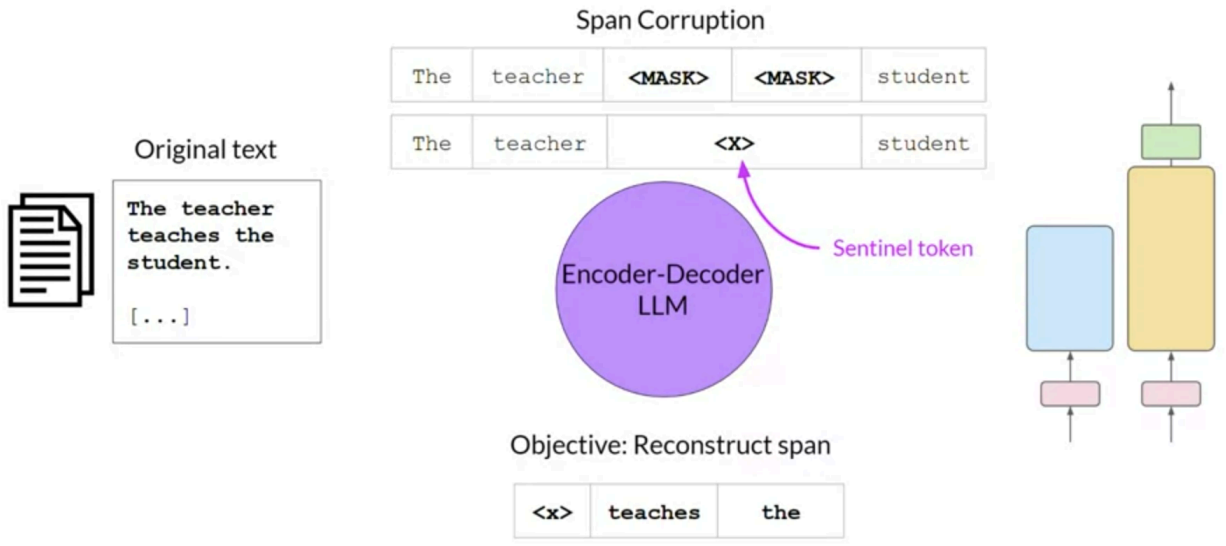
Decoder-only:

Auto-regressive models:
Predicting the next token using the previous sequence. So, the words that are later to the respective token are masked. So, unidirectional context. These are often decoder-only

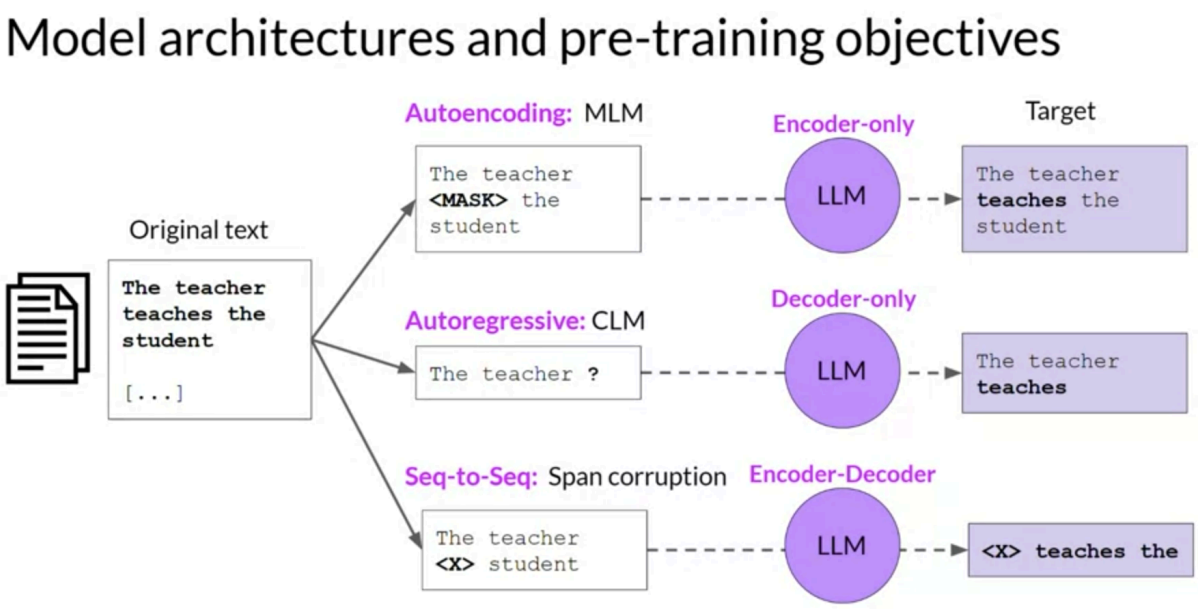


Good use cases: text generation, other emergent behaviour
Examples: GPT, BLOOM

Seq-to-seq models:
Both Encoder and decoder



Example tasks: Text summarization, translation, question answering
Ex: T5, BART



Computational challenges:

Cuda out of memory

We typically need 4 times the memory that we need just to save the model

Quantization

BFLOAT16 has been a popular choice of alternative to FP16 -> not well suited for integer calculations. Full dynamic range of FP32 with half the memory

Quantization: Summary

	Bits	Exponent	Fraction	Memory needed to store one value
FP32	32	8	23	4 bytes
FP16	16	5	10	2 bytes
BFLOAT16	16	8	7	2 bytes
INT8	8	-/-	7	1 byte

FLAN T5

- Reduce required memory to store and train models
- Projects original 32-bit floating point numbers into lower precision spaces
- Quantization-aware training (QAT) learns the quantization scaling factors during training

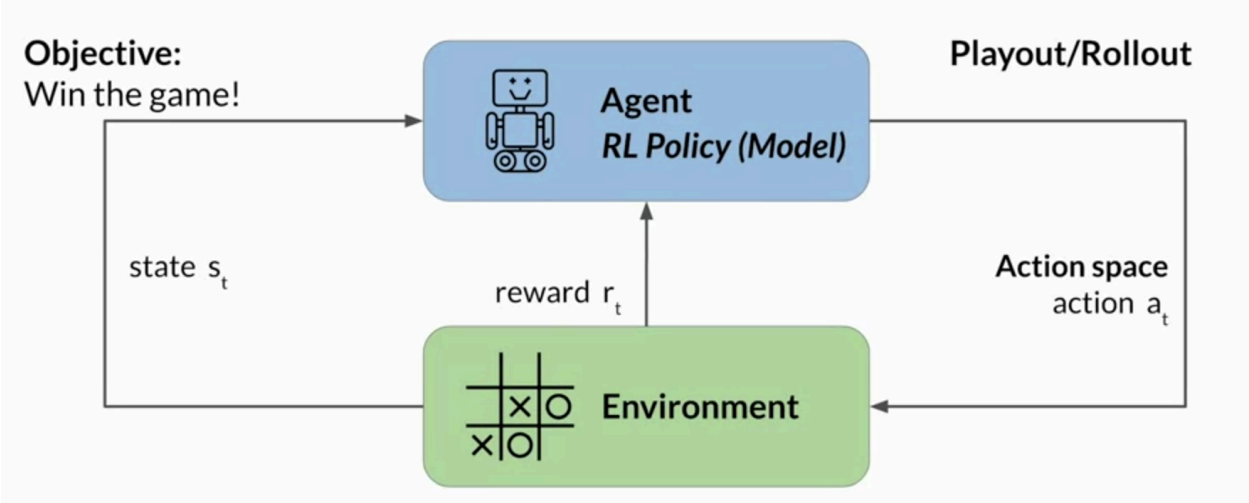
Goal during pre-training -> maximize model performance

Instruction finetuning:

RLHF:

Maximize helpfulness, minimize harm and avoid dangerous topics

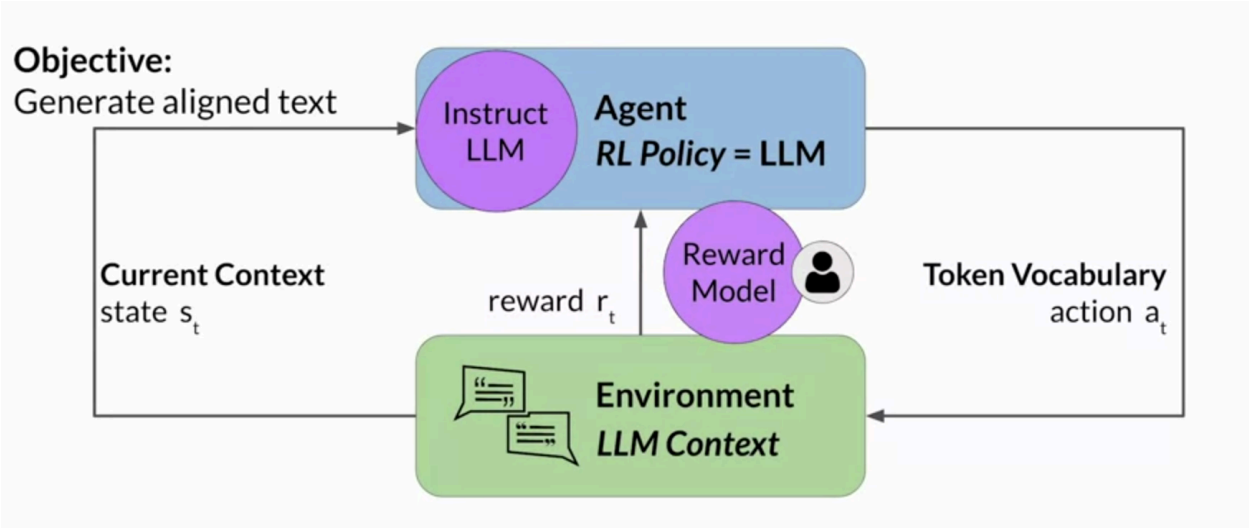
RL:



Agent takes an action by trial and error. Its interaction with the environment results in a change of state of the environment. Then, a reward/penalty is assigned to the agent considering

whether the change of state caused by its action, has taken it closer to the goal state -> Winning the game.

How do we leverage this to fine-tune LLMs?



In this case, the agent's policy that guides the actions is the LLM, and its objective is to generate text that is perceived as being aligned with the human preferences. This could mean that the text is, for example, helpful, accurate, and non-toxic.

The environment is the context window of the model, the space in which text can be entered via a prompt.

The state that the model considers before taking an action is the current context. That means any text currently contained in the context window.

The action here is the act of generating text. This could be a single word, a sentence, or a longer form text, depending on the task specified by the user.

The action space is the token vocabulary, meaning all the possible tokens that the model can choose from to generate the completion.

How an LLM decides to generate the next token in a sequence, depends on the statistical representation of language that it learned during its training. At any given moment, the action that the model will take, meaning which token it will choose next, depends on the prompt text in the context and the probability distribution over the vocabulary space.

The reward is assigned based on how closely the completions align with human preferences. Given the variation in human responses to language, determining the reward is more complicated than in the Tic-Tac-Toe example. One way you can do this is to have a human evaluate all of the completions of the model against some alignment metric, such as determining whether the generated text is toxic or non-toxic. This feedback can be represented as a scalar value, either a zero or a one. The LLM weights are then updated iteratively to maximize the reward obtained from

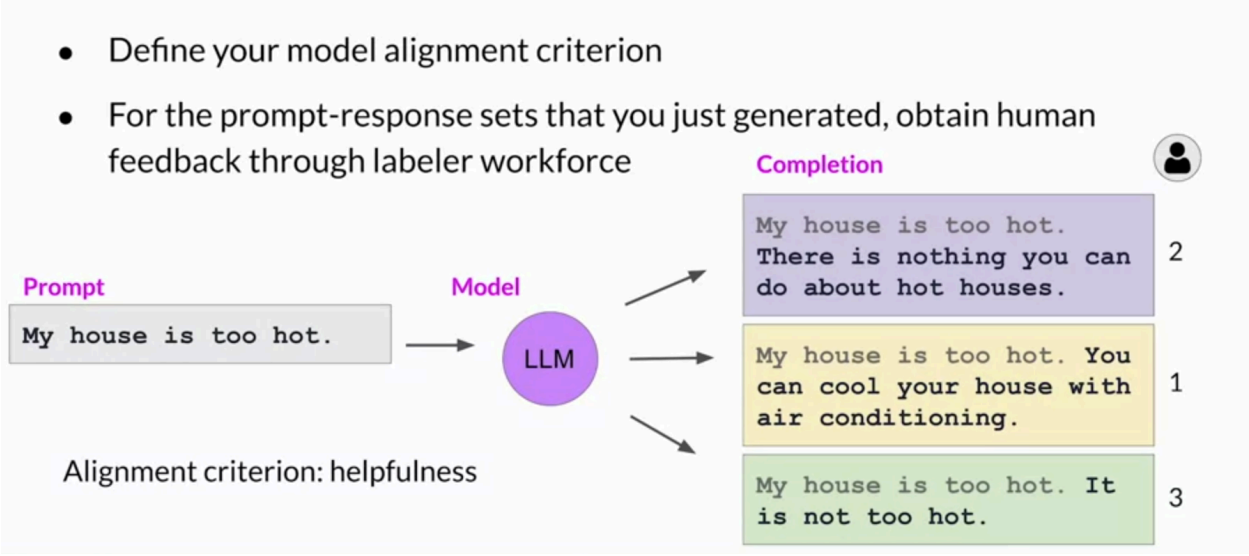
the human classifier, enabling the model to generate non-toxic completions. However, obtaining human feedback can be time consuming and expensive. As a practical and scalable alternative, you can use an additional model, known as the reward model, to classify the outputs of the LLM and evaluate the degree of alignment with human preferences. You'll start with a smaller number of human examples to train the secondary model by your traditional supervised learning methods. Once trained, you'll use the reward model to assess the output of the LLM and assign a reward value, which in turn gets used to update the weights off the LLM and train a new human aligned version. Exactly how the weights get updated as the model completions are assessed, depends on the algorithm used to optimize the policy.

Lastly, note that in the context of language modeling, the sequence of actions and states is called a rollout, instead of the term playout that's used in classic reinforcement learning. The reward model is the central component of the reinforcement learning process. It encodes all of the preferences that have been learned from human feedback, and it plays a central role in how the model updates its weights over many iterations. In the next video, you'll see how this model is trained and how you use it to classify the model's outputs during the reinforcement learning process. Let's move on and take a look..

Process:

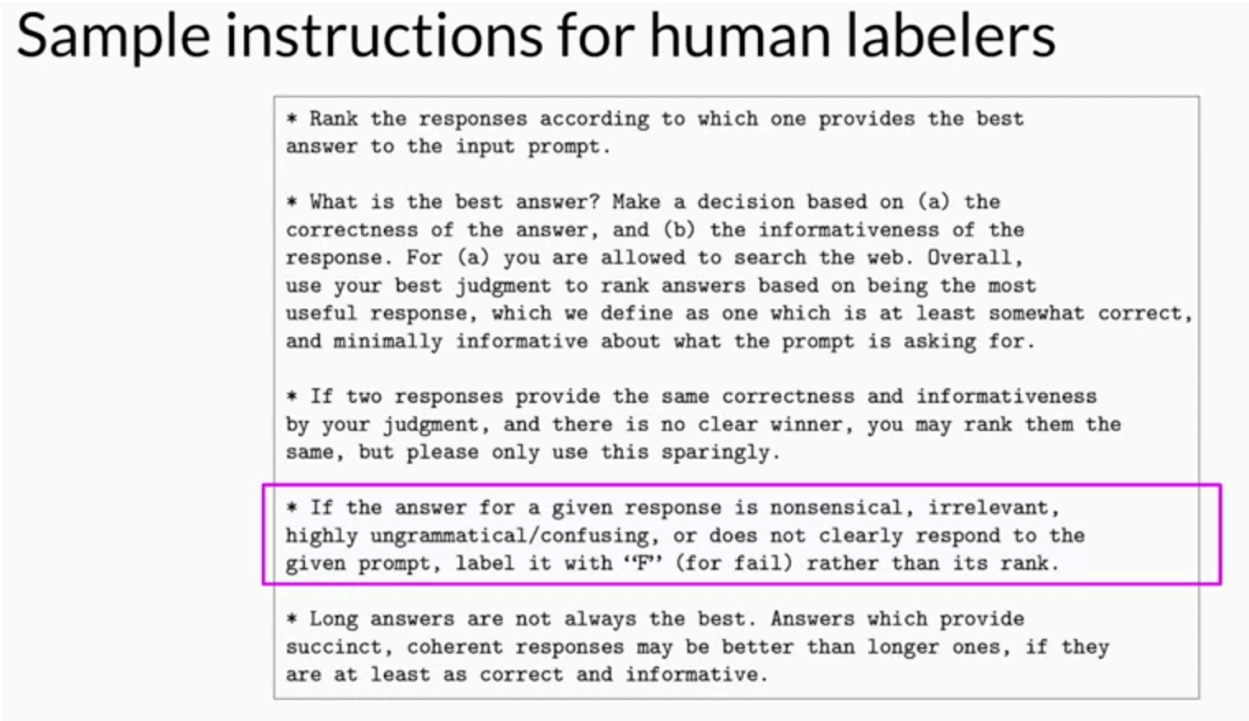
Instruct LLM -> already has a certain ability to follow the task in need

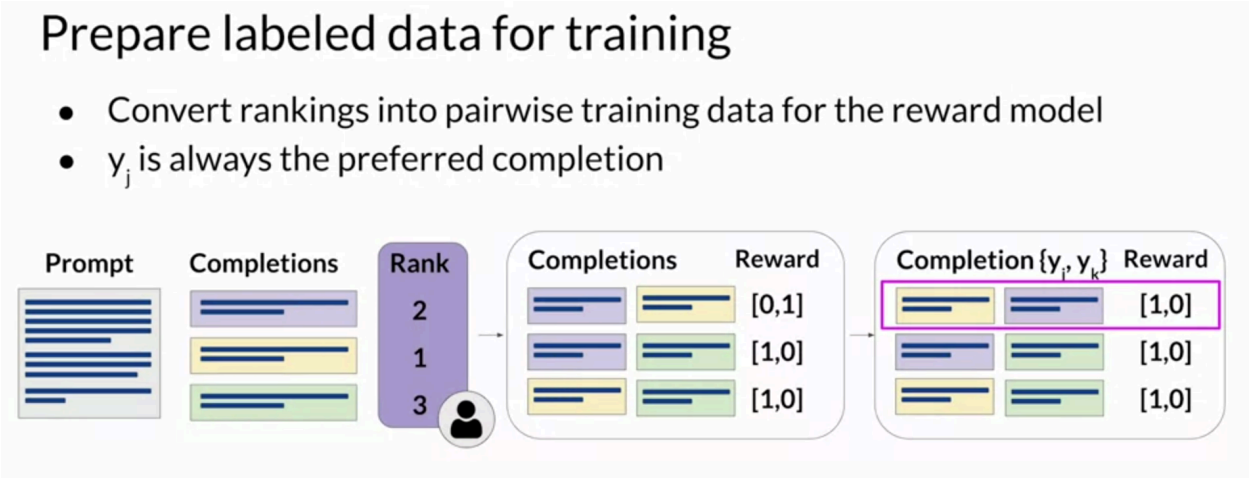
Collect human feedback -> Alignment criterion must be defined properly and then we ask the humans to rate the outputs of the LLM model in accordance with Alignment criterion. (Helpfulness, harm etc)



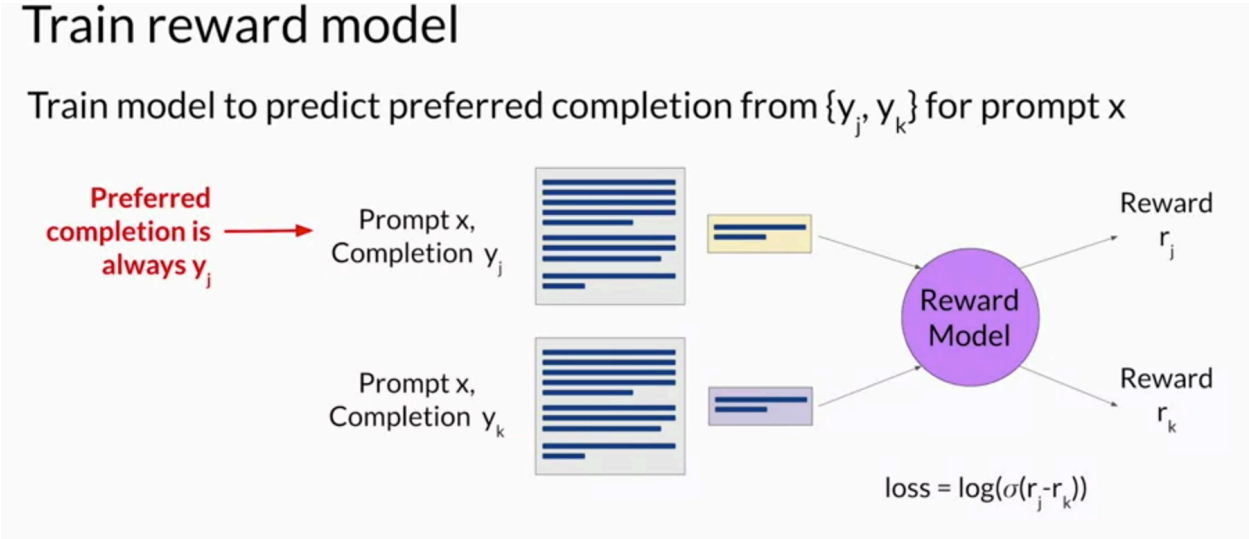
We can use this information from humans and use this as a dataset to train a reward model to generate the appropriate reward function. Typically in the above step, we use multiple humans

to do the same to suppress the noise from unclear alignment criterion or improper human behavior. The sample instructions for human labellers must be very clear.

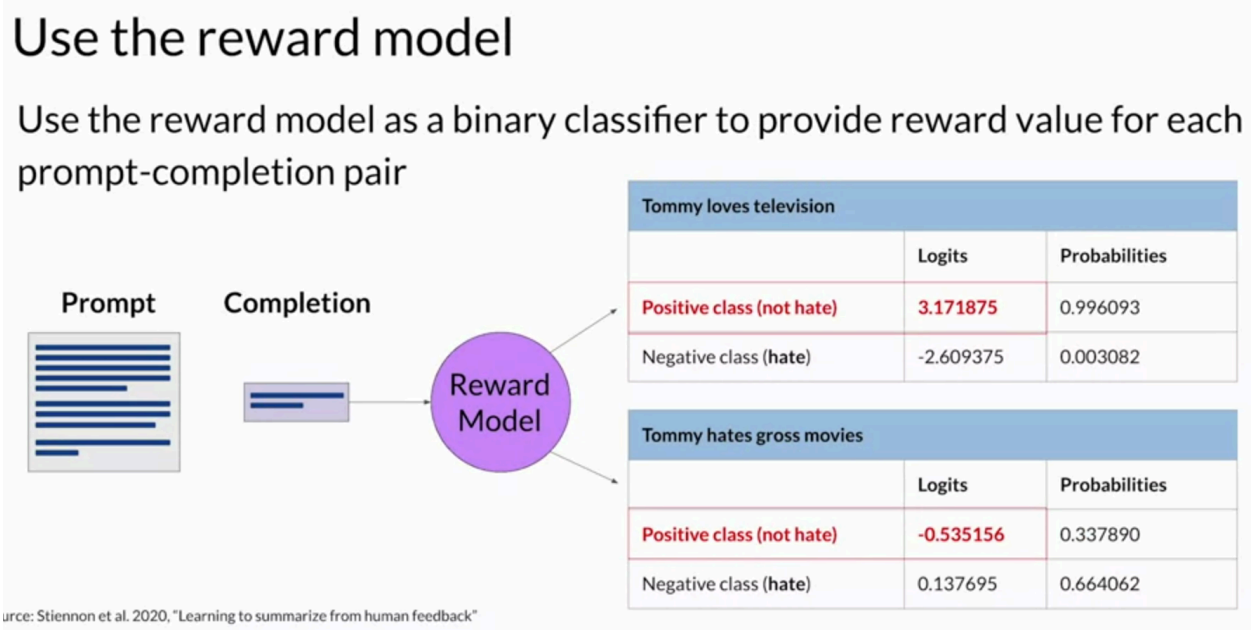




We need to first convert the ranking data into pairwise data. In the above example, there are three different completions for the task and they are ranked according to the alignment criterion by humans. Now, we can make **$^3C_2$ pairs** from the three completion texts. We can have a small reward vector for each pair showing what is the higher ranked and preferred response and what is the lower ranked prompt. Then, we reorder the responses such that the preferred response comes first. And then this data is used to train the reward model.



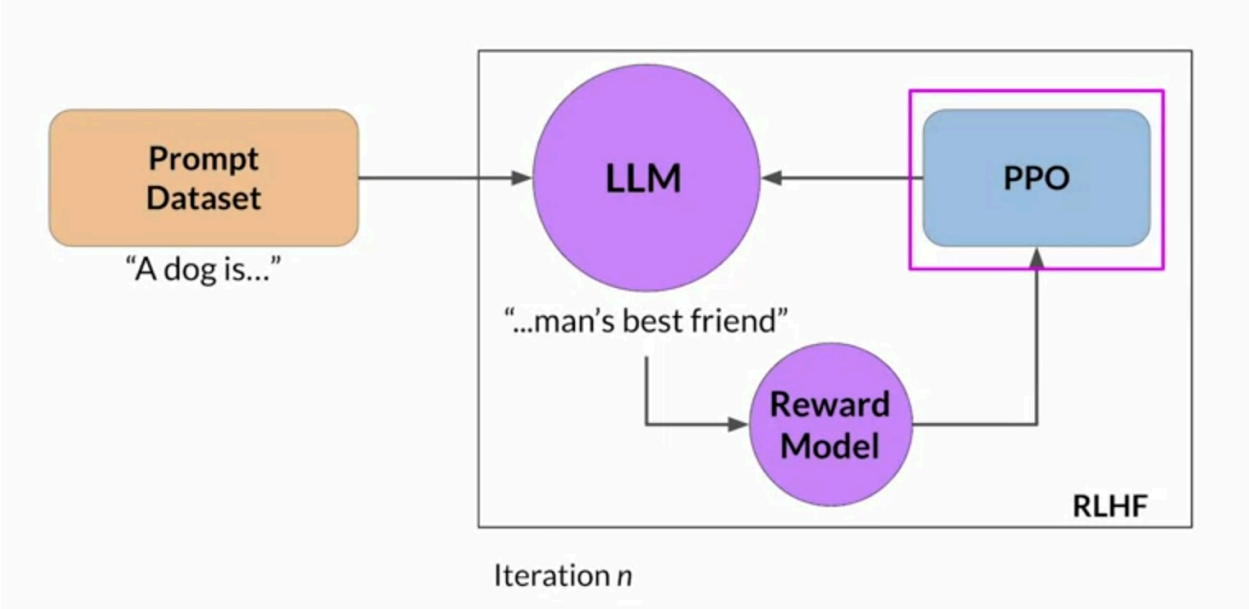
Once we are done training the reward model, we can take humans out of the loop and it can automatically give the right reward and choose the right response. This reward model is usually also a language model that is trained using supervised learning methods on the pair-wise data generated in the above fashion.



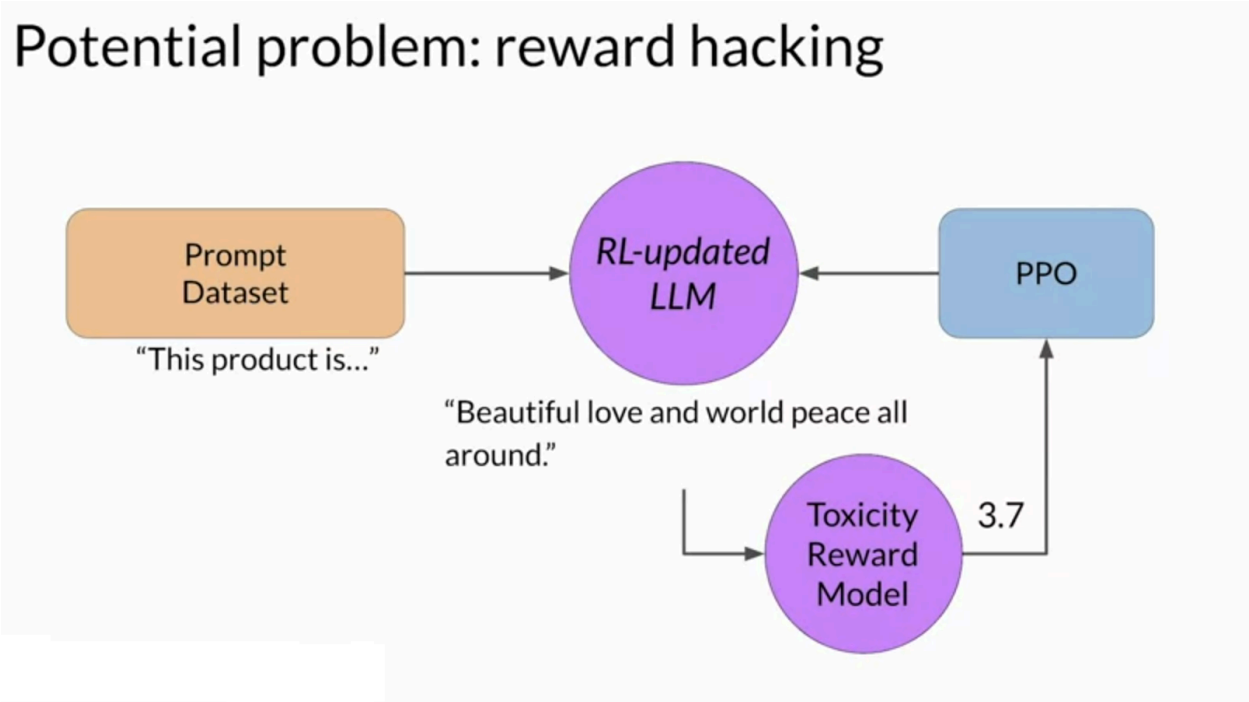
Once the reward model has been trained, we can use the reward model to select the right response in the following way.

Let us say, we need to detoxify our LLM. We need to identify if the completion contains hate speech. After we train our reward model using the datasets created by human labellers, we then use this reward model on a completion. Assume that the reward model has given logits for positive(the one we want to optimize for) and negative classes from the completion and if we

apply softmax to these logits, we get probabilities and hence, we can classify the completion into one out of two categories.



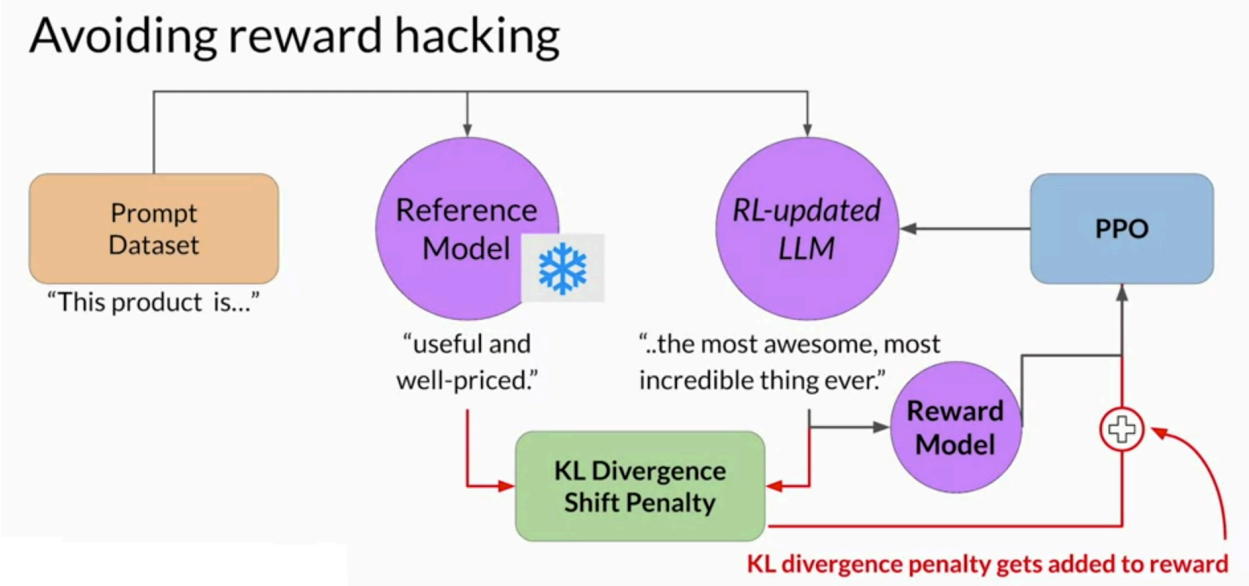
Reward hacking -> Sometimes, optimizing for a reward model can take us far from the direction we want to be according to the actual language model.



Looking for a way to optimize for our reward model, the model might output grammatically incorrect or factually incorrect responses.

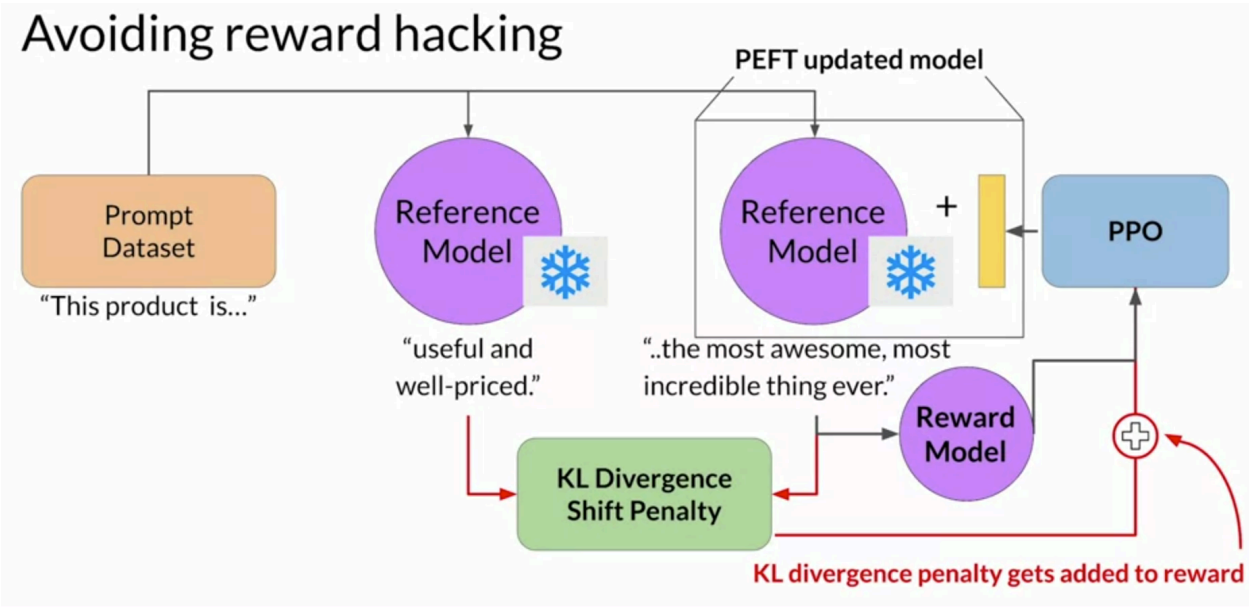
How to prevent Reward Hacking?

The weights of the initial Instruct LLM reference model are frozen and model after getting updated are stored in a separate instance. Now, these two are used to make completion inferences and calculate a quantity called KL Divergence Shift penalty. This shows us if we are being too far away from the actual model.



We add this to the reward and this penalizes the completions if the KL divergence is too huge.

We can also combine with PeFT and update only certain weights of the model. This decreases the memory footprint. (RLHF+PEFT)



We can evaluate the human-aligned LLM with scores on another dataset. We can understand the effect of Human-alignment on the given model.